Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.

HyperLoom Possibilities for Executing Scientific Workflows on the Cloud

Vojtech Cima[1], Stanislav Böhm[1], Jan Martinovič[1], Jiří Dvorský[1], Thomas J. Ashby[2], and Vladimir Chupakhin[3]

[1] IT4Innovations, VŠB Technical University of Ostrava, Ostrava, Czech Republic, firstname.lastname@vsb.cz
[2] IMEC, Brussels, Belgium, ashby@imec.be
[3] Janssen Pharmaceutica NV, Brussels, Belgium, vchupakh@its.jnj.com

Abstract. We have developed HyperLoom - a platform for defining and executing scientific workflows in large-scale HPC systems. The computational tasks in such workflows often have non-trivial dependency patterns, unknown execution time and unknown sizes of generated outputs. HyperLoom enables to efficiently execute the workflows respecting task requirements and cluster resources agnostically to the shape or size of the workflow. Although HPC infrastructures provide an unbeatable performance, they may be unavailable or too expensive especially for small to medium workloads. Moreover, for some workloads, due to HPCs not very flexible resource allocation policy, the system energy efficiency may not be optimal at some stages of the execution. In contrast, current public cloud providers such as Amazon, Google or Exoscale allow users a comfortable and elastic way of deploying, scaling and disposing a virtualized cluster of almost any size. In this paper, we describe HyperLoom virtualization and evaluate its performance in a virtualized environment using workflows of various shapes and sizes. Finally, we discuss the Hyperloom potential for its expansion to cloud environments.

Keywords: cloud, virtualization, distributed environments, scientific workflows, hpc

1 Introduction

The rapid growth of resource demanding workloads such as machine learning applications which take advantage of large-scale infrastructures is being reflected in service offerings of major public cloud providers. For example, Amazon's AWS [1] offers instance types with up to 16 CPUs (64 vCPUs) and hundreds GB of RAM, similarly Exoscale [4] offers instance types with up to 16 CPUs and 128GB RAM. The performance of the compute instances with such specifications is directly comparable with compute nodes in HPC systems. But unlike the HPC systems, these can be deployed on demand in just a few seconds in seemingly any imaginable scale. As the gap between HPC systems and other virtualized distributed environments such as clouds decreases, more HPC solutions are being ported to the cloud.

Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.

Although the trend in Cloud Computing is to shift workloads to lighter virtualization solutions such as containers, the virtualization overhead is still present. Nonetheless, for many use cases is the overhead either acceptable, insignificant or compensated by other factors.

In this paper, we describe a virtualization of HyperLoom - originally an HPC solution for defining and executing scientific workflows and compare the performance of the virtualized HyperLoom to the performance of the bare metal deployment.

The paper is organized as follows. Section 2 describes important properties of scientific workflows. Section 3 introduces HyperLoom - an HPC solution for defining and executing such workflows. In Section 4, we overview standard virtualization solutions and discuss their suitability for virtualization of HyperLoom's components. We evaluate HyperLoom performance in Section 5. Finally, we conclude in Section 6.

2 Scientific workflows

Many scientific workloads are composed of several consecutive computational phases. These are then often chained into more complex flows, such as, for example, model cross-validation combined with hyper-parameter search, which results in pipelines in a shape of large directed acyclic computational graphs - *plans*, whose nodes represent computational units - *tasks*. Figure 1 shows an example of such a graph.

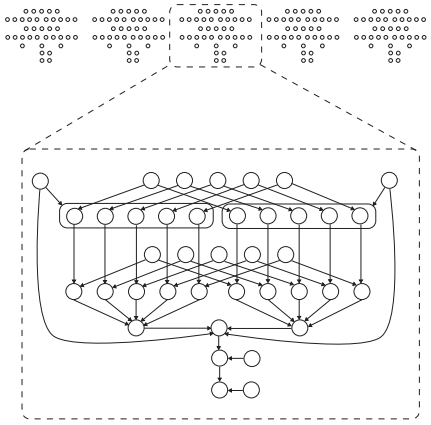


Fig. 1: Example of a scientific pipeline visualized as a directed acyclic graph.

Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.

2.1 Workflow properties

We have identified a set of properties which, we believe, apply to the most of the scientific workflows. Different scientific workloads are defined by plans of various shapes and sizes. Plans may contain millions of tasks of the various types with non-trivial inter-task dependencies. Generally, a plan can take the shape of any directed acyclic graph. Furthermore, the execution time of individual tasks is typically not known before the execution finishes and may vary from milliseconds (short-running tasks) to days (long-running tasks). Similarly, the size of the outputs produced by the tasks may not be known in advance. Distributed environments, namely HPC clusters, may contain thousands of computational nodes. Moreover, different computational nodes may provide various resource types with different capacities. All of the listed properties have a significant impact on the workflow execution. Ideally, we look for a solution that minimizes the execution time agnostically to those properties.

3 HyperLoom

HyperLoom is a platform for defining and executing scientific workflows in distributed environments designed for HPC systems. The ultimate goal of HyperLoom is to minimize the overall plan execution time respecting tasks' and environment's resource constraints.

Other tools for scheduling tasks in distributed environments exist such as Spark [16], HTCondor [11] or Hadoop [14]. If we only consider solutions that allow defining inter-task dependencies, we name SciLuigi [12], DAGman [8], Pegasus [9] and Dask/Distributed [13]. The first three are designed for more coarse-grain tasks. Moreover, in these tools, the inter-task data transfer is done through a shared file system which introduces another performance bottleneck for some use cases, namely in the scenarios with a large number of tasks where a large number of files to be created imposes a significant load on the distributed file system. Also, in all of the cases, the tools do not provide an easy way of chaining third party applications and provide so an arbitrary functionality.

To mitigate the limitations of the competing solutions while respecting the properties listed in Section 2.1, HyperLoom contains the following design features. The core of HyperLoom is implemented in C++. Plans can be easily defined and executed using the client application implemented in Python. Since the execution time of individual tasks is not known in advance, Hyperlooom implements an optimized dynamic scheduler that schedules the tasks reactively with a low overhead. Moreover, the scheduler respects task dependencies and prioritizes placements that induce the smallest possible inter-node data transfer. The data produced by tasks are by default kept directly in memory and can be accessed by any other task from any other node directly with no additional overhead imposed on the server or the underlying file system. HyperLoom allows chaining and execution of third-party applications in the same manner as any other native task type. It is also possible to define custom task types.

Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.

3.1 Architecture

Figure 2 illustrates the main components of HyperLoom. HyperLoom consists of a *server* process that manages *worker* processes that run on computational nodes and a *client* component providing an user interface to HyperLoom.

Client allows users to programmatically chain computational tasks into a plan and submit the plan to server. It also provides a functionality to gather results of the submitted tasks after computation finishes. **Server** receives and decomposes a plan and reactively schedules tasks to run on available computational resources provided by workers. **Workers** execute and run tasks as scheduled by server and inform the server about the state of task execution. This modular architecture allows connecting an arbitrary number of workers which is the keystone for HyperLoom scalability.

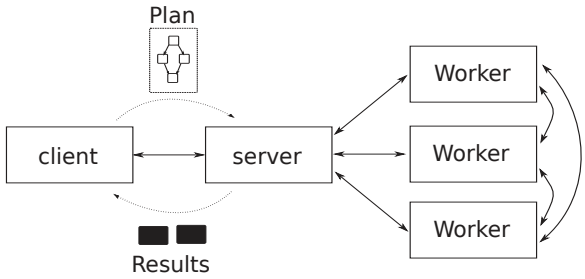


Fig. 2: HyperLoom architecture.

4 Virtualization

In this section, we introduce the main motivation for the virtualization of the HyperLoom infrastructure, overview well-known virtualization solutions and discuss the potential of their usage in the HyperLoom context.

4.1 Motivation for HyperLoom virtualization

HyperLoom has been designed as a high-performance solution in the context of HPC systems. Nonetheless, we realize some of the disadvantages of such systems such as inflexible resource management, limited offerings and high administration overhead comparing to the cloud solutions. We foresee that the HyperLoom virtualization is a key step for its expansion to the cloud world that will allow on-demand and flexible deployment of HyperLoom at infrastructures other than HPC which may be easier to access for some of the potential users.

Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.

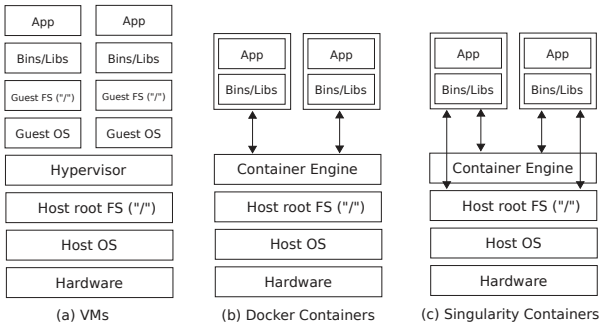


Fig. 3: Virtualization solution: (a) virtual machines, (b) Docker containers, (c) Singularity containers.

4.2 Virtualization solutions

One of the best-known virtualization solutions used nowadays is the concept of virtual machines (VMs) powered by various hypervisors. While the VMs dominated the market not so many years ago, many workloads are being shifted to lighter virtualization platforms such as containers. The adoption of container platforms such as Docker [2] is reportingly increasing. Containers, in contrast to heavier VMs, allow almost an instant execution of a containerized application or a service while still providing a certain level of isolation from the host machine. This shift allows developers to build and ship more flexible and scalable applications available as-a-service. Although Docker has been proven numerous times to work well in cloud environments and provide a centralized catalog - Docker Hub [3] containing many ready-to-run Docker images, the Docker daemon process requires a privileged user (root) to run which makes it unlikely to be widely adopted by HPC centers. Singularity [5], in contrast to Docker, is a containerization solution that does not require the daemon process being owned by root which makes it a containerization solution of choice for many HPC systems. Moreover, Singularity is compatible with existing Docker images including those in Docker Hub. Figure 3 illustrates the main architectural differences between (a) a standard virtual machine, (b) a Docker container and (c) a Singularity container.

4.3 HyperLoom virtualization

We have virtualized the key components of HyperLoom - *server* and *worker* using Ubuntu 16.04 as the base Docker image for creating a container with HyperLoom binaries and the underlying library stack. Although we have only containerized HyperLoom using Docker image, we don't foresee any potential issues creating a HyperLoom virtual machine which would provide the same functionality with higher virtualization overhead. We have decided to use containers mainly due

Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.

to the lighter virtualization layer sacrificing the full process isolation in favor of higher performance and also due to their increasing popularity in the community.

5 Performance evaluation

In this section, we discuss HyperLoom performance. Concretely, we compare plan execution time with and without virtualization scaling the environments up to 64 compute nodes on which we execute various synthetic and a real test case scenarios.

5.1 Testbed description

We have carried out all the experiments on a testbed with up to 64 identical physical computational nodes, each with two 12-core Intel Xeon E5-2680v3 processors (2.5GHz) [6] and 128 GB of physical memory. All the nodes are interconnected by an InfiniBand[15] network (56 Gbps). All nodes run Red Hat Enterprise Linux [10] 6.5 OS. As a virtualization layer, we use Singularity Launcher with HyperLoom components containerized using a Docker container based on Ubuntu 16.04 image. The virtualized deployment is depicted in Figure 4.

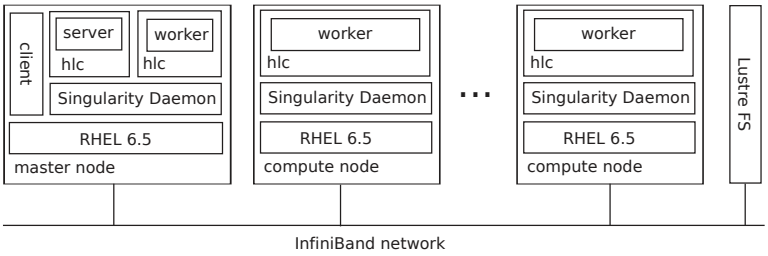


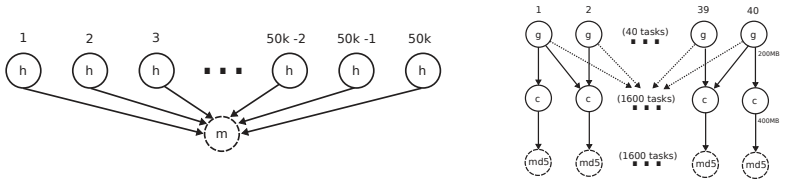
Fig. 4: Virtualized HyperLoom infrastructure.

5.2 Methodology and metrics

We measure the total execution time (execution duration) from the time when a plan is submitted to the server to the time when the computation finishes. To illustrate the virtualization overhead, we also compute and plot the relative execution time by normalizing the plan execution time in virtualized environment by the execution time of the native execution. The test scenarios are described more in detail in Section 5.3 below. All experiments were replicated three times and the total execution time averaged to moderate potential unforeseen system deviation which might affect the performance.

Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.

5.3 Test scenarios



(a) 50kh

(b) gridcat

Fig. 5: Visualization of the (a) *50kh* and (b) *gridcat* test case scenarios.

We have designed several test case scenarios devoted to evaluate different corner cases as well as to evaluate HyperLoom performance and scalability for workflows used in practice. As an example of real-world scenario, we demonstrate a performance of a scientific workflow derived from a pipeline used for novel drug discovery. Below, we briefly describe our test cases more in detail.

50kh - a synthetic test case designed to benchmark scheduling overhead. As depicted in Figure 5a, the assembled plan contains 50k independent and identical short running tasks - *h*-nodes (running *hostname* command) followed by a task that merges outputs from all of them - *m*-node.

gridcat - a synthetic test case designed to simulate more complex workflows. The assembled plan contains tasks of various sizes chained together, so it induces a significant inter-worker data transfer when scheduled inappropriately. Concretely, as visualized in Figure 5b, we create 40 tasks which each generate a 200MB output, followed by tasks representing a concatenation of every possible pair from the first layer resulting in 1600 tasks (each 400MB of output), the last layer of tasks then computes the *md5* hash of the concatenated data.

mlchemo - a real-world test case derived from an existing scientific workflow which performs a nested cross-validation (5×5) with hyperparameter search to find an optimal parametrization of machine-learning based models used for compound activity prediction. The plan contains a mix of long-running tasks such as modeling and validation done by LibSVM [7] - a widely used support vector machine implementation, short running tasks for supporting tasks such us averaging values and others. The shape of this plan is very similar to the plan depicted in Figure 1.

5.4 Experiments

Table 1 contains total execution time values [s] for each of the test case scenarios using both, the virtualized (sing.) and the native (native) HyperLoom deployment. The total execution time for both of the synthetic test cases (*50kh*, *gridcat*)

Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.

Table 1: Comparison of the plan execution time [s] using native and virtualized HyperLoom deployment (*50kh*, *gridcat* *mlchemo*).

	50kh		*gridcat*		*mlchemo*	
# nodes	native	sing.	native	sing.	native	sing.
1	48.84	44.37	112.08	115.03	28,796.97	31257.13
2	23.41	20.97	76.00	83.53	14123.65	15,279.46
4	12.19	10.56	49.05	48.80	7,025.58	7,599.67
8	7.48	6.38	35.37	35.75	3,547.31	3,851.36
16	10.09	9.89	34.85	42.49	1,815.60	1,989.85
24	13.59	14.53	31.94	48.78	1,255.27	1,370.00
32	18.11	16.80	38.96	37.01	956.81	1,053.71
64	34.28	33.99	50.68	43.59	559.27	595.12

varies from seconds to minutes depending on the cluster size. We remind that these test cases were designed to artificially stress HyperLoom components with the increasing cluster size and thus we do not observe the inverse proportionality between the cluster size and the total execution time values. The scalability of Loom is more evident in the *mlchemo* test case, where the total execution time steadily decreases with the increasing cluster size.

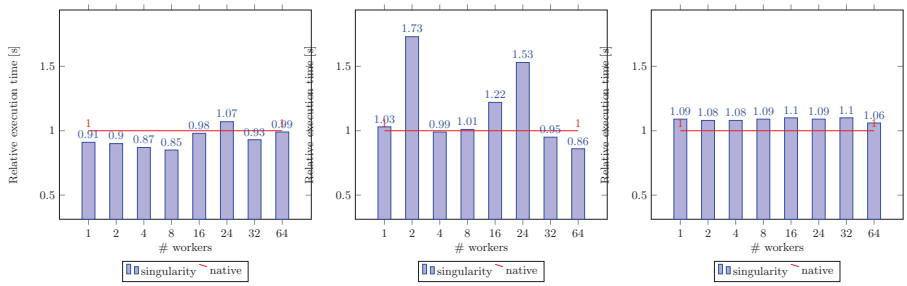
Figure 6 visualizes the virtualization overhead by plotting the relative execution time for the respective test case scenarios.

None of the synthetic test cases - *50kh* (Figure 6a) nor *gridcat* have confidently confirmed the expected performance degradation caused by the virtualization layer. On the contrary, for the *gridcat* test case, the virtualized deployment of HyperLoom performed slightly better than the native deployment in 7 out of 8 cases (1, 2, 4, 8, 16, 32, 64 workers) with $\sigma = 0.069$. The *gridcat* test case (Figure 6b) introduces significantly higher variance ($\sigma = 0.193$) in the relative execution time of the virtualized deployment with varying cluster size. In this test case, the virtualized infrastructure performed slightly better in 3 out of 8 cases (4, 32, 64 workers).

A representative of real-world HyperLoom usage - the *mlchemo* test case (Figure 6c) uniformly confirmed the expected performance degradation caused by the added virtualization layer. The virtualized deployment of HyperLoom is in this case in average 9% slower than the native alternative. Notably, in this case, we observe a very low variation ($\sigma = 0.01$) in the relative execution time values with varying cluster size.

Considering the non-trivial size and complexity of the *mlchemo* test case, whose execution time varied from more than 8 hours (1 worker) to less than 20 minutes (64 workers), we argue that these results may be generalized to other large-scale real-world workloads more objectively than the results obtained using the synthetic test cases.

Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.



(a) *50kh* (b) *gridcat* (c) *mlhemo*

Fig. 6: Relative execution time of the (a) *50kh*, (b) *gridcat* and (c) *mlchemo* test cases executed on virtualized infrastructure (singularity) normalized to the execution time of the test cases executed on native HyperLoom infrastructure (native baseline) using up to 64 nodes (24 CPUs each).

6 Conclusion

We have successfully virtualized HyperLoom components using Docker images with Singularity Launcher and compared the performance of such a virtualized infrastructure to native HyperLoom deployment in an HPC environment.

Despite the fact that HyperLoom was initially designed for HPC systems, we have shown its potential of being used in a cloud or other virtualized environments which brings HyperLoom closer to a much broader audience. Although cloud providers offer a flexible lease of machines which performance is comparable to those in HPC systems, the network solutions used in HPC systems offer incomparably higher inter-node throughput and latency.

We have shown that the degradation of HyperLoom performance caused by the virtualization layer is in average $\sim$9% for the test case designed to simulate real-world scenarios. Nevertheless, some of the synthetic test cases in virtualized environment slightly outperformed the performance of the bare metal HyperLoom deployment. This suggests that advantages of virtualization such as the availability of the newest versions of application dependencies may outweigh the overhead of virtualization layer itself.

Although we have virtualized HyperLoom so it can find its audience outside the HPC community, we believe that virtualization is also beneficial for HPC systems as it has the potential to enable more applications on such systems without the need for the applications themselves being directly compatible with the underlying operating systems.

Although we have only discussed fix-sized HyperLoom deployments, in elastic cloud environments, we foresee a potential for HyperLoom to become significantly much more energy efficient by scaling the deployment up/down (potentially in/out) based on current workers utilization which is one of the subjects for future work.

Complex, intelligent, and software intensive systems, 2017. New York, NY: Springer Berlin Heidelberg. ISBN 978-3-319-61565-3.

7 Acknowledgements

This project has received funding from the European Unions Horizon 2020 Research and Innovation programme under Grant Agreement No. 671555. This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science - LQ1602 and by the IT4Innovations infrastructure which is supported from the Large Infrastructures for Research, Experimental Development and Innovations project IT4Innovations National Supercomputing Center LM2015070.

References

1. Amazon AWS. "https://aws.amazon.com/".
2. Docker. "https://www.docker.com/".
3. Docker Hub. "https://hub.docker.com/".
4. Exoscale. "https://www.exoscale.ch/".
5. Singularity. "http://singularity.lbl.gov/".
6. Specsheet - Processor Intel Xeon E5 2680. "http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz".
7. Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.
8. Weiwei Chen and Ewa Deelman. Workflow overhead analysis and optimizations. In *Proceedings of the 6th Workshop on Workflows in Support of Large-scale Science*, WORKS '11, pages 11–20, New York, NY, USA, 2011. ACM.
9. Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005.
10. Red Hat. Red hat enterprise linux. "https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux", 2017. [Online; accessed 31-March-2017].
11. HTCondor. Htcondor. "https://research.cs.wisc.edu/htcondor/index.html", 2017. [Online; accessed 31-March-2017].
12. Samuel Lampa, Jonathan Alvarsson, and Ola Spjuth. Towards agile large-scale predictive modelling in drug discovery with flow-based programming design principles. *Journal of Cheminformatics*, 8(1):67, 2016.
13. Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, pages 130–136. Citeseer, 2015.
14. Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.
15. Wikipedia. Infiniband — wikipedia, the free encyclopedia. "https://en.wikipedia.org/w/index.php?title=InfiniBand&oldid=772443735", 2017. [Online; accessed 31-March-2017].
16. Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.